A linked data framework for Android

Maria Rosoiu, Jérôme David, Jérôme Euzenat

A linked data framework for Android

Maria-Elena Roşoiu, Jérôme David, Jérôme Euzenat

INRIA & Univ. Grenoble Alpes
Grenoble, France
{Maria.Rosoiu,Jerome.David,Jerome.Euzenat}@inria.fr

Abstract. Mobile devices are becoming major repositories of personal information. Still, they do not provide a uniform manner to deal with data from both inside and outside the device. Linked data provides a uniform interface to access structured interconnected data over the web. Hence, exposing mobile phone information as linked data would improve the usability of such information. We present an API that provides data access in RDF, both within mobile devices and from the outside world. This API is based on the Android content provider API which is designed to share data across Android applications. Moreover, it introduces a transparent URI dereferencing scheme, exposing content outside of the device. As a consequence, any application may access data as linked data without any a priori knowledge of the data source.

1 Introduction

Smartphones are becoming our main personal information repositories. Unfortunately, this information is stored in independent silos managed by applications, thus it is difficult to share data across them. One could synchronize application data, such as the contacts or the agenda using a central repository. However, these are not generic solutions and there is no mean to give access to data straight from the phone. The W3C Device API[1] covers this need across devices, but it offers specific APIs for specific applications, and not a uniform and flexible access to linked data. Nowadays, mobile operating systems, such as Android, deliver solutions to access application content, but they are restricted to some application database schemas that must be known beforehand.

Offering phone information as RDF data would allow application developers to take advantage of it without relying on specific services. Moreover, doing this along the linked data principles (use URIs, provide RDF, describe in ontologies, link to other sources) would integrate the phone information within the web of data and make it accessible from outside the phone. Our goal is to provide applications with a generic layer for data delivery in RDF. Using such a solution, applications can exploit device information in an uniform way without knowing, from the beginning, application schemas.

For example, an application may be used as a personal assistant: when one would like to know which of his contacts will participate to an event, he will consult the calendar of his contacts in order to retrieve the answer: are they participating to the event or will they be around the place? From data in linked data and the guest food preferences,

[1] `http://www.w3.org/2007/uwa/Activity.html`

it should be possible to select suitable nearby restaurants. Finally, from guest availability and restaurant opening hours, it could adequately plan for a meeting and deliver an invitation to these people. For sure, privacy and security concerns will have to be dealt with appropriately, but in a first step we are concerned by making these data available and interoperable.

The Android platform has several appealing features for that purpose:

- Applications are built and communicate in a service oriented architecture;
- Data sharing is built-in through the notion of ContentProviders.

We presented a first version of RDF content providers in [2]. This layer, built on top of Android content providers, allowed to share application data inside phones. In this paper, we extend it by adding capabilities to access external RDF data, and to share application data as linked data on the web. The mobile device information can then be accessed remotely, from any web browser, by any person who has been granted access to it. In this case, the device acts like a web server.

Early efforts were made to build an homogeneous XML repository from personal information [14]. Some pioneering attempts at marrying mobile information and semantic web technologies were made in [7], but do not aim to expose RDF data. The Nepomuk project[2] strived to produce RDF PIM ontologies, and to expose desktop data in RDF. OinkIt [6] exported phone contacts as FOAF files. OinkIt is restricted to contacts though Nepomuk covers a wide variety of PIM applications. Nepomuk replicates data in an RDF store and OinkIt generates a file, while we would rather only provide access to this data.

This paper is a comprehensive presentation of a framework for exposing Android application data as linked data. We first describe the context in which the Android platform stores its data (§2), and how it can be extended in order to integrate RDF (§3). Then, we present three types of applications that sustain its feasibility (§4): the first type of applications wrap several facilities of devices to expose their data as linked data, the second application allows one to annotate pictures stored inside the phone, and the last one is an RDF browser that acts like a linked data client. We then explain the behavior of a server which exposes phone information to the outside world (§5). We discuss some technical issues raised by this framework and the solutions we implemented for them (§6). Finally, we present future improvements and challenges in this field (§7).

2 The Android architecture

Android is a Linux-based operating system for mobile phones. It allows for developing applications in Java [8, 5] based on a service oriented architecture that we present here.

2.1 Services and Intents

Android is built around different kinds of components that are provided by an application[3]:

[2] `http://nepomuk.semanticdesktop.org/`

[3] `http://developer.android.com/guide/topics/fundamentals.html`

– Activities are user interaction modalities (an application panel, a chooser, an alert);
– Services are processing tasks;
– Broadcast receivers are components that react to events;
– Content providers expose some data of an application to other applications.

An application implements any of these kinds of components. Applications and application components communicate through messages called "intent(s)". They are defined as:

```
Intent intent = new Intent( Action, Data );
```

such that Action is a Java like package name, e.g., `fr.inrialpes.exmo.rdfoid.GETRDF`, and Data can be anything, but would generally be a URI, e.g., `content://contacts/people/33`, and optionally a mime-type specifying the expected result. The intent must be called through:

```
startService( intent ).
```

The targeted component can be explicit, i.e., the components to deal with the request are explicitly identified, or Android can look for an application or component able to answer the Action on the Data, and pass it the call and the arguments.

2.2 Android Content Providers

Inside the Android system, each application runs in isolation from other applications. For each application, the system assigns a different and unique user. Only this user is granted access to the application data. This allows one to take advantage of a secure environment, however this tends to lock data in independent repositories, each of them with its own data representation. This prevents data sharing across applications.

Content providers overcome this drawback by enabling the transfer of structured data between device applications through a standard interface. This interface empowers one to query the data or to modify it.

A content provider[4] is a subclass of `ContentProvider` and implements the following interface:

```
Cursor query( Uri id, String[] proj, String select, String[] selectArgs, String orderBy )
Uri insert( Uri id, ContentValues colValueList)
int update( Uri id, ContentValues colValueList, String select, String[] selectArgs )
int delete( Uri id, String select, String[] selectArgs )
String getType( Uri id ) .
```

which allows one to query, to insert, to delete or to update the data. Queries are issued in an SQL manner and the results are returned as a cursor on a table.

Calling a `ContentProvider` is driven by the kind of content to be manipulated: the calling application indicates its desire to retrieve some content through its type and/or URI, but does not control which application will provide it. Android calls a `ContentResolver` which further looks into the query (the `id`) to find a suitable content provider on the phone for providing the required content. For that purpose, the resolver maps the query URIs to the declared providers. These providers are declared in application manifest file.

[4] `http://developer.android.com/guide/topics/providers/content-providers.html`

2.3 Android URIs

Android URIs have a specific structure:

```
content://authority/path/to/data/optionalID.
```

The `content` scheme indicates that the identified resource is delivered from a content provider, the `authority` identifies the provider, the `path/to/data` identifies a particular table, and the `optionalID` distinguishes a particular instance in the table, like in:

```
content://contacts/people/33;
content://fr.inrialpes.exmo.pikoid/picture/234.
```

The URI `content://contacts/people` refers to all the people in the contact application, and the URI `content://contacts/people/33` identifies a specific instance of these, namely the instance having the id 33.

When an application requires access to a particular piece of data, it queries its URI. This is done through a request addressed to the `ContentResolver` which routes the query to the corresponding content provider.

The usage of URIs to identify data is a key strength from a linked data standpoint. However, these URIs use the specific `content` protocol instead of HTTP, and content providers do not return RDF.

Moreover, the URIs used by content providers are local to each device, i.e., not dereferenceable on the web, and not unique. This constraint is adequate only when the interface is used within the same Android device. However, when the interface connects to a wider context, this constraint does not maintain its validity: the `content://contacts/people/22` refers to different entries, i.e., contacts, stored inside the phone book agenda of different mobile devices.

3 The RDF Content Provider Framework

To allow Android applications to exchange RDF data, we need `ContentProvider`s which deliver their data in RDF. For that purpose, we have designed an API which must be embedded inside applications that offer or access RDF content.

3.1 Framework overview

In order to achieve this, we have primarily followed the same principles as the ones used in the original `ContentProvider` API. Therefore, our `RDFContentProvider` API delivers the following classes and interfaces:

- `RdfContentProvider`: An abstract class that should be extended if one wants to create an RDF content provider. It subclasses the `ContentProvider` class belonging to the Android framework;

– `RdfContentResolverProxy`: A proxy used by applications to send queries to the `RDFContentResolver` application. The `RDFContentResolver` application records all the RDF content providers installed on the device and routes queries to the relevant provider;
– `Statement`: A class used for representing RDF statements;
– `RdfCursor`: An iterator on a set of RDF statements;
– `RdfContentProviderWrapper`: A subclass of `RdfContentProvider` which allows for adding RDF content provider capabilities to an existing classical content provider.

Figure 1 gives an overview of the framework architecture.

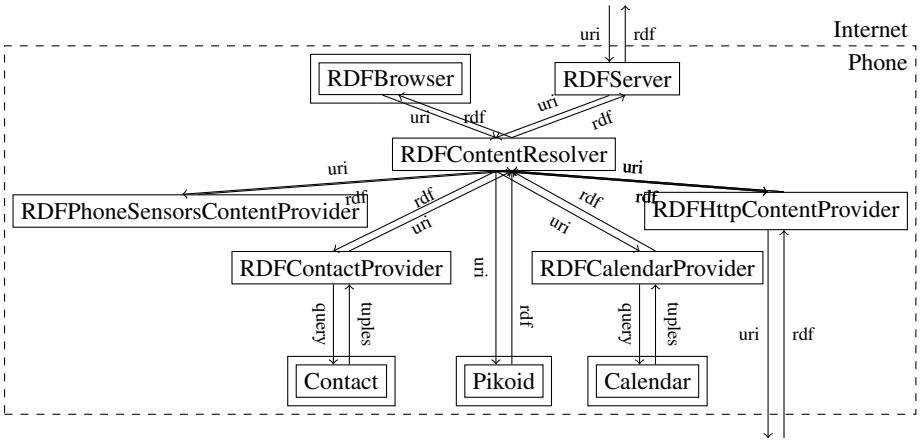

Fig. 1. The architecture components and the communication between them. Components with double square have a relevant graphic user interface.

The main components from a developer perspective are the `RDFContentProvider` API and the `RDFContentResolver` application.

3.2 The RDF Content Provider API

The goal of the `RDFContentProvider` API is to answer to two types of queries:

– Queries that request information about a particular individual, e.g., tell me what you know about contact 33. The provided answer is a set of triples which corresponds to the description of one object and its attribute values.
– Queries that request only the values for some variables that must satisfy a specific condition, i.e., SPARQL-like queries. In this case, the answer is a table of tuples, like in `ContentProvider`s or SPARQL.

For the first type of queries we provide a minimal interface. This interface has to be implemented for linked data applications and has the following format:

– `RDFCursor getRdf( Uri id ).`

In this case, the cursor iterates on a table of subject-predicate-object (or predicate-object) which represents the triples involved in the description of the object given as a URI.

As for the second type of queries, they require a more elaborate semantic web interface, i.e., a minimal SPARQL endpoint. Thus, the following methods have to be also implemented:

- `Uri[] getTypes( Uri id )`: returns the RDF types of a local URI;
- `Uri[] getOntologies()`: ontologies used by the provider;
- `Uri[] getQueryEntities()`: classes and relations that can be delivered by the provider;
- `Cursor query( SparqlQuery query )`: returns tuple results;
- `Cursor getQueries()`: triple patterns that can be answered by the provider.

The `RDFContentProvider`s that we have developed so far only implement the first three primitives.

This interface corresponds to the one we required to web services in our work on ambient intelligence [4]. Indeed, to some extent this work is similar to the work in ambient intelligence except that instead of working in a building-like environment, it works within the palm of one's hand. But the problem is the same: applications which do not know each others can communicate through semantic web technologies.

3.3 The RDF Content Resolver Service

The `RDFContentResolver` service has the same goal as the `ContentResolver` belonging to the Android framework. It maintains the list of all installed `RDFContentProviders`, and forwards the queries it receives to the corresponding ones. Users do not have to interact with this application, therefore it is implemented as an Android service.

When an RDF content provider is instantiated by the system, a principle similar to the one from the Android content provider framework is used: this provider automatically registers to the `RDFContentResolver` (Figure 2).

The `RDFContentResolver` can route both the local (`content:`) and external (`http:`) URI-based queries. In case of a local URI, i.e., starting with the `content` scheme, the resolver decides to which provider it must redirect the query. In case of an external URI, i.e., starting with the `http` scheme, the provider automatically routes the query to the `RDFHttpContentProvider` (see Figure 1).

4 Applications

We developed a few applications as a proof of concept of this framework. They can be considered in different categories: wrappers for existing Android content providers and other services (§4.1), native applications (§4.2) and a client example for the framework (§4.3).

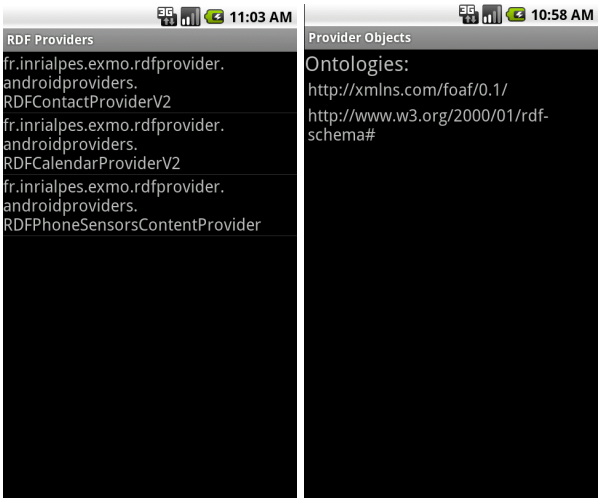

Fig. 2. RDFBrowser also allows for inspecting from the RDFContentResolver available RDFContentProviders and the ontologies they manipulate.

4.1 RDF Provider wrappers for Phone applications

The `RDFContentResolver` application is also bundled with several RDF content providers encapsulating the access to Android predefined providers. The Android framework has applications which can manage the address book and the agenda. These two applications store their data inside their own content provider.

In order to expose this data as RDF, we developed the `RDFContactProvider` and the `RDFCalendarProvider`. These providers are wrapper classes for the `ContactProvider` and the `CalendarProvider` residing inside the Android framework. The same could be obtained by bypassing the `ContentProvider` interface, and using instead the W3C Device APIs since they exist for each of these applications[5]. So doing should not be significantly more difficult and would ease porting to other platforms than Android.

`RDFContactProvider` exposes contact data using the FOAF ontology (see Figure 2). It provides data about the name of a person (display name, given name, family name), his phone number, email address, instant messenger identifiers, homepage and notes.

`RDFCalendarProvider` provides access to Android calendar using the RDF Calendar ontology[6]. The data supplied by this provider is information about events, their location, their date (starting date, ending date, duration, and event time zone), the organizer of the event and a short description.

In addition to these content providers, two other RDF providers are the `RDFPhoneSensorsContentProvider` and `RDFHttpContentProvider`.

[5] `http://www.w3.org/2009/dap/` or `http://www.w3.org/TR/geolocation-API/`

[6] RDF Calendar vocabulary: `http://www.w3.org/TR/rdfcal/`.

`RDFPhoneSensorsContentProvider` exposes sensor data from the sensors embedded inside the mobile device. Contrary to the others, they are not offered as content providers. At the present time, it only delivers the geographical position (retrieved using the Android LocationManager service). In order to express this information in RDF, we use the geo location vocabulary[7], which provides a namespace for representing lat(itude) and long(itude).

The `RDFHttpContentProvider` allows one to retrieve RDF data from the web of data. It parses RDF documents retrieved by dereferencing URIs through HTTP and presents them as `RDFCursor`s. So far, only the minimal interface has been implemented, i.e., the `getRdf( Uri id )` method.

Developing a wrapper would consist, in general, of the following steps:

- Identify data exposed in the application content provider;
- Choose ontologies corresponding to this data;
- Provide a URI pattern for each ontology concept;
- Implement a dereferencing mechanism which, for each type of resource, extracts information from the content provider and generates RDF from this (generating URIs for related resources).

A native RDF content provider application may follow the same steps. However, it may be developed without any content provider. In this case, the analysis has to be carried out from the application data (or a corresponding API).

4.2 Pikoid

Pikoid is a native implementation of an RDF content resolver, i.e., an application that directly implements this interface.

It is a simple application allowing users to annotate pictures on the phone. The annotations answer the following simple questions: where and when (the picture was taken), who (is on the picture) and what (it represents). It is strongly integrated in the Android platform as it uses other content providers for identifying these annotations: people are taken from the address book, places from the map and events from the calendar.

Pikoid directly provides access to this data in RDF: each pikoid object offers these annotations as well as reference to the corresponding objects served by the wrapped content providers (`RDFContactProvider` and `RDFCalendarProvider`). Figure 3 illustrates browsing starting from Pikoid.

4.3 RDF Browser

The RDF Browser acts as a linked data client. Given a URI, either `http:` or `content:`, the browser issues a request to the `ContentResolver`. It then displays the resulting RDF cursor content as a simple page. If the data contains other URIs, the user can click on one of them and the browser will issue a new query with the selected URI.

An example can be found in Figure 4. In this case, the user uses the `RDFBrowser` to get information about the contact having the id 4. When the browser receives the

[7] Geo location vocabulary: `http://www.w3.org/2003/01/geo/`.

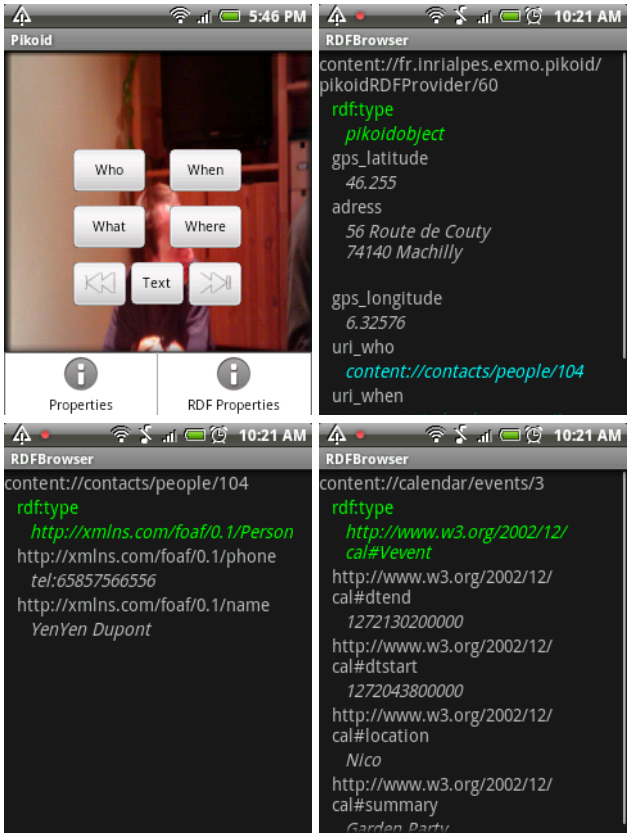

Fig. 3. The Pikoid application annotates images with metadata stored as RDF. RDFBrowser allows for querying this information to the Pikoid RDFContentProvider interface and displaying it. The current picture metadata is shown in the second panel (pikoidRDFprovider/60). From there, it is possible to browse the information available in the address book (people/104) and the calendar (events/3) through the corresponding RDF content providers wrapping them.

request, it sends it further to the `RDFContentResolver`. Since the URI starts with the `content://` scheme and has the `com.android.contacts` authority, the resolver routes the query to the `RDFContactProvider`. This provider retrieves the set of triples describing the contact and sends it to the calling application which displays it to the user. Thereupon, the user decides that he wants to continue browsing and selects the homepage of the contact. In this case, since the URI starts with the `http://` scheme, the resolver routes the query to the `RDFHttpContentProvider`. The same process repeats and the user can see the remote requested file, i.e., Tim Berners-Lee FOAF file.

5 RDF Server: embedding a phone in the web of data

`RDFContentProvider`s serve linked data within a single phone, `RDFServer` exposes this linked data to the wider web of data. This is based on three components:

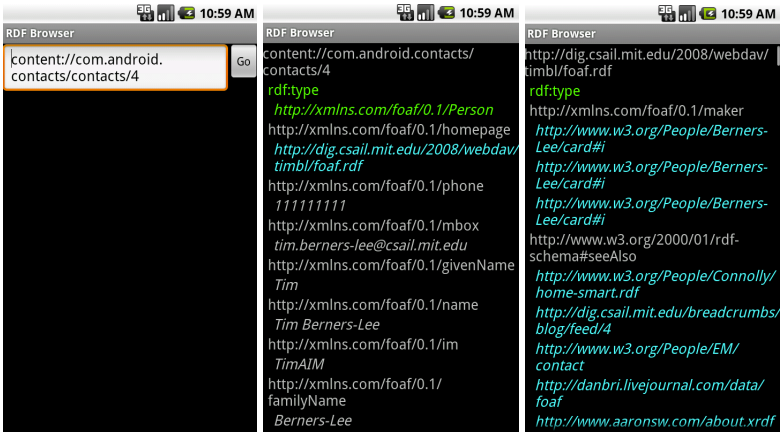

Fig. 4. An example of using the RDF Browser for accessing remote RDF.

- `RDFServer` is an RDF HTTP server that takes incoming HTTP queries (URIs) and returns RDF;
- the `RDFContentResolver` dereferences incoming URIs and externalizes local URIs within RDF;
- `RDFHttpContentProvider` allows for following HTTP URIs if necessary (see Figure 1).

5.1 RDF Server

The `RDFServer` exposes the data stored into the device as RDF to the outside world. Because the server must permanently listen for new requests and does not require any user interaction, it is implemented as an Android service, i.e., a background process.

The server principles are quite simple. At launch time, it listens on port 80 for incoming requests. Once it receives a request from the outside, it dereferences the requested URI, i.e., it translates the external URI into an internal one, which has a meaning inside the Android platform. The `RDFServer` sends it further to the `RDFContentResolver`. In a manner similar to the one explained for the `RDFBrowser`, the set of triples is obtained. Before sending this set to the server, the URIs of the triples are externalized, i.e., transformed into `http:` URIs. The graph is then serialized using a port of Jena under the Android platform.

5.2 Dereferencing and externalising URIs

One important issue appears when one want to get data from a device because the URIs used to query the content providers have a local meaning. URIs used to query the address book of two different devices are the same, but the content it identifies will likely be different.

The URI externalization process translates the local URI:

```
content://authority/path/to/data
```

into the dereferenceable one:

`http://deviceIPAddress:port/authority/path/to/data.`

Reversing the translation of such a URI is possible since both the authority and the path are preserved by the externalization process.

Usually, mobile devices do not have a permanent IP address and thus, the externalized URIs are not stable. To overcome this, a dynamic DNS client[8][9] may be used.

In addition, the server supports a minimal content negotiation mechanism. If one wants to receive the data in RDF/XML, it will set the MIME types of the Accept-type header of its request to "application/rdf+xml" or to "application/*". In the opposite case or when the client sets the MIME type to "text/plain", the data will be transmitted in the N-Triple format. Not only the requester has the opportunity to express its preferences regarding the format of the received data, but the default format of the transmitted data can be specified in the server settings, as well the port on which the server can listen on and the domain name server for it.

Fig. 5. RDF Server response from externalized URIs.

An example can be found in Figure 5. In this scenario, the user retrieves information about the fourth contact from the device address book. The request is processed by the RDF Server in a manner similar to the one of the RDF Browser.

6 Technical issues: application size

The `RDFServer` included in our architecture eases the access of the user to the RDF data found on the web. For that purpose, we wanted to reuse an existing semantic web framework, such as Jena or Sesame. Yet these are not suitable to be employed under the Android platform (the code depends on some libraries that are unavailable under Android). There are a few ports of these frameworks to Android: Microjena[10] and An-

[8] Dynamic DNS Client: `https://market.android.com/details?id=org.l6n.dyndns&hl=en`.

[9] DynDNS: `http://dyn.com/dns/`.

[10] `http://poseidon.ws.dei.polimi.it/ca/?page_id=59`.

drojena[11] are ports of Jena and there exists a port of Sesame to the Android platform mentioned in [1]. We use Androjena.

A problem that arises when we use this framework is that the size of the application increases substantially. This is one of the constraints identified in [11]. This problem could have been avoided by reimplementing only the Jena modules that are needed in our architecture. Still, we would like to improve our architecture by adding more features (such as a SPARQL query engine) that require additional modules to those used to read/parse/write RDF, available in Jena.

We have used ProGuard for addressing this problem. ProGuard[12] is a code shrinker, optimizer, and obfuscator. It removes the unused classes, methods or variables, performs some byte-code optimizations and obfuscates the code. We only took advantage of the two former features. The tool proved to be efficient in reducing the size of our application (our framework including Androjena) by half, i.e., its initial size was 6.48MB, and, after applying ProGuard, it was reduced to 3.15MB. In Table 1, the effect of ProGuard can be observed on some of the applications that we developed.

Application	Size without ProGuard	Size with ProGuard
RDFContentResolver	6.49 MB	3.15 MB
RDFBrowser	368 KB	184 KB
RDFServer	100 KB	76 KB
Alignment API impl.	254 KB	170 KB

Table 1. Size of applications with or without ProGuard.

The existence of such tools as ProGuard, is a step forward in the continuous battle between applications that require a considerable amount of space for storing their code and devices with a reduced memory storage.

7 Perspectives

The current framework is only a first step towards a more comprehensive semantization of Android devices. Here are some further steps that we plan to take.

7.1 SPARQL querying

One of these further steps would be to allow one to query the device data using SPARQL.

A double problem appears when one would like to achieve this: the distribution of the query across several content providers and its translation.

The distribution will require query partitioning and dispatching to different providers as performed in distributed query processing [13, 10].

The translation of the query can be addressed in several manners:

[11] `http://code.google.com/p/androjena/`.

[12] `http://proguard.sourceforge.net/`.

– creating a new RDF content provider which relies on a triple store to deposit the data [9], and then using SPARQL to query it;
– translating SPARQL queries into more specific requests that may be answered by an RDF content provider;
– translating SPARQL queries into SQL queries and further decompose them into ones compatible with the ContentProvider interface.

Concerning the second option, there are several available tools that can make the translation from SPARQL to SQL, like Virtuoso or D2RQ. However, these tools solve only half of the problem because the SQL queries have to be adapted to the Content-Provider interface, i.e., the queries have a particular format, different from SQL. This interface allows for querying only one view of a specified table at a time, hence it is not possible to ask content providers to perform joins.

7.2 Query mediation

Once one is able to query data, the heterogeneity of the ontology used by providers may be a problem. Overcoming this requires mediating queries, i.e., transforming query expressed into one ontology in another query expressed with an ontology understood by a content provider. For that purpose, we plan to use ontology alignments. We already provide a micro version of the Alignment API[13] [3] working under Android and able to retrieve alignments from an Alignment server.

7.3 Security and privacy

Challenges regarding security must be taken into account. The user of the application should be able to grant or to deny access to his personal data. A specific vocabulary, such as the one introduced in [12], should be used in order to express this. Moreover, the dangers of granting system access to a third-party user can be avoided by using a secure authentication protocol[14].

7.4 Resource consumption

Finally, the problem of resource consumption is mentioned here for the record. Such resources may be related to bandwidth (WiFi or 3G) that are consumed by having the RDFServer working. In addition, such a server, and the use of our framework in general, may affect energy consumption. This will have to be precisely considered.

8 Conclusion

Involving Android devices in the semantic web, both as consumers and providers of data, is an interesting challenge. As mentioned, it faces the issues of size of applications

[13] `http://alignapi.gforge.inria.fr`
[14] `http://www.w3.org/wiki/WebAccessControl` and `http://www.w3.org/wiki/Foaf+ssl`.

and URI dereferencing in mobility situations. There remain other technical problems in implementing a full Android RDF framework encompassing distributed SPARQL querying.

So, our next step is to provide a more fine grained and structured access to data through SPARQL querying. This promises to raise the issue of computation, and thus energy, cost on mobile platform.

A further issue is the control of privacy in such a framework. In this particular domain too, we think that semantic technologies can provide more flexible and targeted solutions.

The framework and applications described here are available at `http://swip.inrialpes.fr`.

Acknowledgements

We thank Yu Yang and Loïc Martin who have programmed part of the Pikoid application. This work has been partially supported by the French National Research Agency under grant ANR-10-CORD-009 (Datalift).

References

1. Mathieu d'Aquin, Andriy Nikolov, and Enrico Motta. Building SPARQL-enabled applications with android devices. In *Proc. 10th ISWC demonstration track, Bonn (DE)*, 2011.
2. Jérôme David and Jérôme Euzenat. Linked data from your pocket: The Android RDFContentProvider. In *Proc. 9th ISWC demonstration track, Shanghai (CN)*, pages 129–132, 2010.
3. Jérôme David, Jérôme Euzenat, François Scharffe, and Cássia Trojahn dos Santos. The Alignment API 4.0. *Semantic web journal*, 2(1):3–10, 2011.
4. Jérôme Euzenat, Jérôme Pierson, and Fano Ramparani. Dynamic context management for pervasive applications. *Knowledge engineering review*, 23(1):21–49, 2008.
5. Marko Gargenta. *Learning Android*. O'Reilly Media, Inc., Sebastopol (CA US), 2011.
6. Ora Lassila. Semantic web approach to personal information management on mobile devices. In *Proc. IEEE International Conference on Semantic Computing (ICSC), Santa Clara (CA US)*, pages 601–607, 2008.
7. Marko Luther, Yusuke Fukazawa, Matthias Wagner, and Shoji Kurakake. Situational reasoning for task-oriented mobile service recommendation. *Knowledge engineering review*, 23(1):7–19, 2008.
8. Reto Meier. *Professional Android 2 Application Development*. Wrox, Birmingham (UK), 2011.
9. Danh Le Phuoc, Josiane Xavier Parreira, Vinny Reynolds, and Manfred Hauswirth. RDF On the Go: An RDF storage and query processor for mobile devices. In *Proc. 9th ISWC demonstration track, Shanghai (CN))*, November 2010.
10. Bastian Quilitz and Ulf Leser. Querying distributed RDF data sources with SPARQL. In *Proc. 5th ESWC, Tenerife (ES)*, pages 524–538, 2008.
11. Laurens Rietveld and Stefan Schlobach. Semantic web in a constrained environment. In *Proc. ESWC Downscaling the semantic web workshop, Heraklion (GR)*, pages 31–38, 2012.
12. Owen Sacco and Alexandre Passant. A privacy preference ontology (PPO) for linked data. In *Proc. WWW Linked Data on the Web Workshop (LDOW2011)*, pages 1–5, 2011.

13. Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. FedX: optimization techniques for federated query processing on linked data. In *Proc. 10th ISWC, Bonn (DE)*, pages 601–616, 2011.
14. Norman Walsh. Generalized metadata in your Palm. In *Proc. 2nd Extreme markup languages conference, Montréal (CA)*, 2002.